PCT

WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : G06F 9/38, 9/45, 11/28	A1	(11) International Publication Number: WO 99/19795 (43) International Publication Date: 22 April 1999 (22.04.99)

(21) International Application Number: PCT/US98/21465

(22) International Filing Date: 9 October 1998 (09.10.98)

(30) Priority Data:

08/953,836	13 October 1997 (13.10.97)	US
09/168,040	7 October 1998 (07.10.98)	US

(71) Applicant: INSTITUTE FOR THE DEVELOPMENT OF EMERGING ARCHITECTURES, L.L.C. [US/US]; c/o Hewlett-Packard Company, Legal Dept. M/S 44L18, 19111 Pruneridge Avenue, Cupertino, CA 95014-0795 (US).

(72) Inventors: MORRIS, Dale, C.; 442 Gilbert Avenue, Menlo Park, CA 94025 (US). MILLS, Jack, D.; 1768 Chevalier Drive, San Jose, CA 95124 (US). CHEN, William, Y.; 1477 Yukon Drive, Sunnyvale, CA 94087 (US).

(74) Agent: HAGGARD, Alan, H.; Hewlett-Packard Company, Legal Dept., M/S 20BN, P.O. Box 10301, Palo Alto, CA 94303-0890 (US).

(81) Designated States: AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CU, CZ, DE, DK, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, UA, UG, UZ, VN, YU, ZW, ARIPO patent (GH, GM, KE, LS, MW, SD, SZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

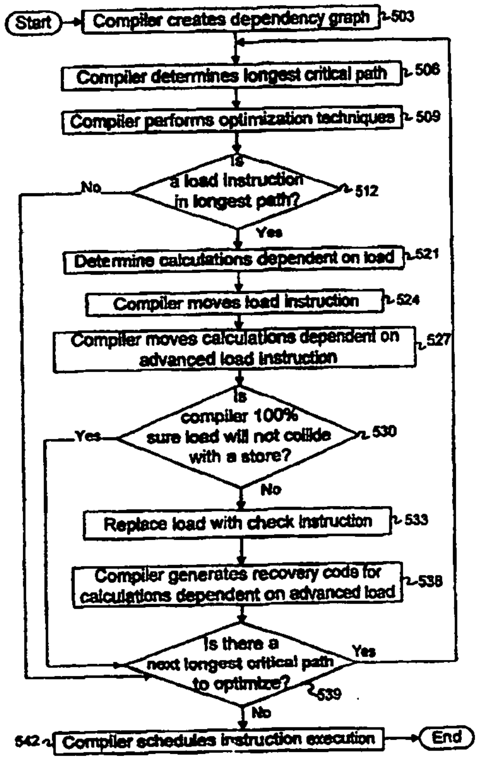
Published

With international search report.
Before the expiration of the time limit for amending the claims and to be republished in the event of the receipt of amendments.

(54) Title: METHOD AND APPARATUS FOR OPTIMIZING INSTRUCTION EXECUTION

(57) Abstract

Problems encountered during speculative execution of instructions are deferred. If the results of instructions that were speculatively executed are subsequently used, the integrity of the execution of the instructions is verified. If not, recovery code is executed that alters the state of the computer system (50) to create the appearance that the speculatively executed instructions had executed successfully.


Start
Compiler creates dependency graph
503
Compiler determines longest critical path
506
Compiler performs optimization techniques
509
Is a load instruction in longest path?
512
No
Yes
Determine calculations dependent on load
521
Compiler moves load instruction
524
Compiler moves calculations dependent on advanced load instruction
527
Is compiler 100% sure load will not collide with a store?
530
Yes
No
Replace load with check instruction
533
Compiler generates recovery code for calculations dependent on advanced load
538
Is there a next longest critical path to optimize?
539
Yes
No
542
Compiler schedules instruction execution
End

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania
AM	Armenia
AT	Austria
AU	Australia
AZ	Azerbaijan
BA	Bosnia and Herzegovina
BB	Barbados
BE	Belgium
BF	Burkina Faso
BG	Bulgaria
BJ	Benin
BR	Brazil
BY	Belarus
CA	Canada
CF	Central African Republic
CG	Congo
CH	Switzerland
CI	Côte d'Ivoire
CM	Cameroon
CN	China
CU	Cuba
CZ	Czech Republic
DE	Germany
DK	Denmark
EE	Estonia
ES	Spain
FI	Finland
FR	France
GA	Gabon
GB	United Kingdom
GE	Georgia
GH	Ghana
GN	Guinea
GR	Greece
HU	Hungary
IE	Ireland
IL	Israel
IS	Iceland
IT	Italy
JP	Japan
KE	Kenya
KG	Kyrgyzstan
KP	Democratic People's Republic of Korea
KR	Republic of Korea
KZ	Kazakstan
LC	Saint Lucia
LI	Liechtenstein
LK	Sri Lanka
LR	Liberia
LS	Lesotho
LT	Lithuania
LU	Luxembourg
LV	Latvia
MC	Monaco
MD	Republic of Moldova
MG	Madagascar
MK	The former Yugoslav Republic of Macedonia
ML	Mali
MN	Mongolia
MR	Mauritania
MW	Malawi
MX	Mexico
NE	Niger
NL	Netherlands
NO	Norway
NZ	New Zealand
PL	Poland
PT	Portugal
RO	Romania
RU	Russian Federation
SD	Sudan
SE	Sweden
SG	Singapore
SI	Slovenia
SK	Slovakia
SN	Senegal
SZ	Swaziland
TD	Chad
TG	Togo
TJ	Tajikistan
TM	Turkmenistan
TR	Turkey
TT	Trinidad and Tobago
UA	Ukraine
UG	Uganda
US	United States of America
UZ	Uzbekistan
VN	Viet Nam
YU	Yugoslavia
ZW	Zimbabwe

10871861-2. Morris, Dale C. et al.

METHOD AND APPARATUS FOR OPTIMIZING INSTRUCTION EXECUTION

Cross Reference to Related Applications

This application is a continuation-in-part of U.S. Patent Application Serial Number 08/953,836 filed on October 13, 1997.

Field of the Invention

The present invention relates to the execution of instructions in computer systems. One aspect of the present invention relates to the recovery of an exception caused by computer instructions that are speculatively executed. Another aspect of the present invention relates to advancing execution of an instruction and calculations dependent thereon out of order to achieve improved performance.

Description of the Related Art

A "basic block" is a contiguous set of instructions bounded by branches and/or branch targets, containing no branches or branch targets. This implies that if any instruction in a basic block is executed, all instructions in the basic block will be executed, i.e., the instructions contained within any basic block are executed on an all-or-nothing basis. The instructions within a basic block are enabled for execution when control is passed to the basic block by an earlier branch targeting the basic block ("targeting" as used here includes both explicit targeting via a taken branch as well as implicit targeting via a not taken branch). The forgoing implies that if control is passed to a basic block, then all instructions in the basic block must be executed; if control is not passed to the basic block, then no instructions in the basic block are executed. The act of executing, or specifying the execution of, an instruction before control has been passed to the instruction is called "speculation." Speculation performed by the processor at program runtime is called "dynamic speculation" while speculation specified by the compiler is called "static speculation." Dynamic speculation is known in the prior art. While the vast majority of the prior art is not based on, and does not refer to, static speculation, recently some references to static speculation have begun to surface.

Two instructions are called "independent" when one does not require the result of the other; when one instruction does require the result of the other the instructions are called "dependent." Independent instructions may be executed in parallel while dependent instructions must be executed in serial fashion. Program performance is improved by identifying independent instructions and executing as many of them in parallel as possible. Experience indicates that more independent instructions can be found by searching across multiple basic blocks than can be found by searching only within individual basic blocks. However, simultaneously executing instructions from multiple basic blocks requires speculation.

Identifying and scheduling independent instructions, and thereby increasing performance, is one of the primary tasks of compilers and processors. The trend in compiler and processor design has been to increase the scope of the search for independent instructions in each successive generation. In prior art instruction sets, an instruction that may generate an exception cannot be speculated by the compiler since, if the instruction causes an exception, the program may exhibit erroneous behavior. This restricts the useful scope of the compiler's search for independent instructions and makes it necessary for speculation to be performed at program runtime by the processor via dynamic speculation. However, dynamic speculation entails a significant amount of hardware complexity that increases exponentially with the number of basic blocks over which dynamic speculation is applied - this places a practical limit on the scope of dynamic speculation. By contrast, the scope over which the compiler can search for independent instructions is much larger - potentially the entire program. Furthermore, once the compiler has been designed to perform static speculation across a single basic block boundary, very little additional complexity is incurred by statically speculating across several basic block boundaries.

If static speculation is to be undertaken, then several problems must be solved, one of the most important of which is the handling of exceptional conditions encountered by statically speculated instructions. Since, as noted above, exceptions on speculative instructions cannot be delivered at the time of execution of the instructions, a compiler-visible mechanism is desired to defer the delivery of the exceptions until control is passed to the basic block from which the instructions were speculated (known as the "originating basic block"). Mechanisms that perform a similar function exist in the prior art for deferring and later delivering exceptions on dynamically speculated instructions. However, by definition the mechanisms are not visible to the compiler and therefore cannot be manipulated by the compiler into playing a role in compiler-directed speculation. No known method or apparatus for deferring and later delivering fatal and non-fatal exceptions on statically speculated instructions has been enabled in the prior art. Limited forms of static speculation do exist in the prior art, however: (1) the forms do not involve deferral and later recovery of exceptional conditions, and (2) the forms do not enable static speculation over the breadth and scope of the present invention.

Therefore, when undertaking static speculation, there is a need in the art for a mechanism to handle exceptions on speculative instructions such that any side effects of the speculative instructions are not visible to the programmer. Further, the mechanism should apply to as many forms of static speculation as possible.

There is also a need for a mechanism to achieve higher performance in computer systems by enabling execution of as many independent instructions in parallel as possible. This is desirable even when there is a possibility that a second instruction, as well as a calculation dependent thereon, may operate upon data that can be dependent upon the execution of a first instruction.

Summary of the Invention

In one illustrative embodiment of the invention, a computer readable medium is provided having a compiled program stored thereon in a computer-readable form. The program includes a store instruction and a load instruction that is scheduled before the store instruction; a calculation instruction that is dependent on data read by the load instruction and is scheduled ahead of the store instruction; and a check instruction to determine whether the store instruction and the load instruction access a common location in memory.

In another illustrative embodiment of the invention, a computer system is provided including a memory, means for executing a store instruction, means for executing a load instruction before the store instruction, and means for executing, before the store instruction, a calculation instruction dependent on data read by the load instruction. The computer system also includes means for determining whether the store instruction and the load instruction accessed a common location in the memory.

In another illustrative embodiment of the invention, a computer system is provided including a compiler to create an execution schedule for a source program that includes a load instruction, a store instruction and a calculation instruction that is dependent on data read by the load instruction. The compiler includes means for scheduling the load and calculation instructions ahead of the store instruction when the compiler cannot be certain that the store and load instructions will not access a common memory location during execution of the program.

In another illustrative embodiment, a computer readable medium having a compiled program stored thereon in a computer-readable form is provided. The program includes a store instruction, a load instruction that is scheduled before the store instruction and a check instruction to determine whether the store instruction and the load instruction access a common memory location during execution of the program. The check instruction changes control flow when the check instruction determines that the store instruction and the load instruction accessed a common memory location.

In another illustrative embodiment, a computer system is provided including a memory, means for checking whether a store instruction and a previously-executed load instruction accessed a common location in the memory and means for changing control flow to recovery code when it is determined that the store instruction and the load instruction accessed a common location in the memory.

In another illustrative embodiment, a computer readable medium is provided that is encoded with a compiler that, when executed on a computer system, performs a method of compiling a source program that includes a load instruction, a store instruction and a calculation instruction that is dependent on data read by the load instruction. The method includes the steps of determining whether the store and load instructions will not access a common memory location during execution of the program, and when it cannot be determined that the store and load instructions will not access a common memory location, scheduling the load instruction and the calculation instruction ahead of the store instruction.

In another illustrative embodiment, a computer readable medium encoded with a compiler is provided that, when executed on a computer system, performs a method of compiling a program that includes a first instruction and a second instruction, wherein the compiler cannot be certain that the second instruction will not operate upon data that is dependent upon the execution of the first instruction. The method includes the steps of scheduling the second instruction ahead of the first instruction and generating a check instruction to, during execution of the program, determine whether the second instruction operates on data that is dependent upon the execution of the first instruction.

In another illustrative embodiment a computer readable medium encoded with a compiler is provided that, when executed on a computer system, performs a method of compiling a source program that includes a load instruction, a store instruction and a calculation instruction that is dependent on data read by the load instruction. The method includes the steps of scheduling the load and calculation instructions ahead of the store instruction and generating a branch instruction that branches when it is determined, during execution of the program, that the store and load instructions accessed a common memory location. The method further includes generating recovery code to which the branch instruction branches when it is determined that the store instruction and the load instruction accessed a common memory location during execution of the program, the recovery code including a copy of the load instruction and the calculation instruction.

Another embodiment illustrative embodiment is directed to a method of executing instructions, comprising executing at least one instruction that has been marked as speculative; verifying integrity of execution of the at least one instruction; when the integrity of execution of the at least one instruction is verified, continuing executing other instructions; and when the integrity of execution of the at least one instruction is not verified, executing recovery code and continuing executing the other instructions after executing the recovery code.

A further illustrative embodiment of the invention is directed to a computer readable medium that is encoded with a compiler that, when executed on a computer system, performs a method of compiling a source program to generate a compiled program that includes a plurality of instructions organized in a plurality of basic blocks, each basic block including a set of contiguous instructions, the plurality of instructions including a first instruction that is associated with a first basic block and that may generate an exception during execution of the compiled program. The method comprises steps of: (A) scheduling the first instruction outside of the first basic block and ahead of at least one instruction that precedes the first basic block; and (B) generating a check instruction to, during execution of the compiled program, determine whether the first instruction generates an exception.

Another illustrative embodiment of the invention is directed to a computer readable medium having a program stored thereon in a computer-readable form. The program comprises a plurality of instructions organized in a plurality of basic blocks, each basic block including a set of contiguous instructions. The plurality of instructions includes a first instruction that is associated with a first basic block and that may generate an exception during execution of the program, wherein the first instruction is scheduled outside of the first basic block and ahead of at least one instruction that precedes the first basic block; and a check instruction to, during execution of the program, determine whether the first instruction generates an exception.

A further illustrative embodiment of the invention is directed to a computer system, comprising means for executing a program including a plurality of instructions organized in a plurality of basic blocks, each basic block including a set of contiguous instructions, the plurality of instructions including a first instruction that is associated with a first basic block and that may generate an exception during execution of the program; means for executing the first instruction outside of the first basic block and ahead of at least one instruction that precedes the first basic block; and means for determining whether the first instruction generates an exception.

Another illustrative embodiment of the invention is directed to a computer system, comprising a compiler to create an execution schedule for a program that includes a plurality of instructions organized in a plurality of basic blocks, each basic block including a set of contiguous instructions, the plurality of instructions including a first instruction that is associated with a first basic block and that may generate an exception during execution of the program. The compiler includes means for scheduling the first instruction outside of the first basic block and ahead of at least one instruction that precedes the first basic block when the compiler cannot be certain that the first instruction will not generate an exception during execution of the program.

A further illustrative embodiment of the invention is directed to a computer readable medium having a program stored thereon in a computer-readable form. The program comprises a first speculative instruction that is capable of experiencing an instruction exception condition during execution of the first speculative instruction, wherein the first speculative instruction defers signaling an instruction exception when the exception condition is initially detected and completes execution without signaling the instruction exception.

Another illustrative embodiment of the invention is directed to a computer readable medium encoded with a compiler that, when executed on a computer system, performs a method of compiling a source program to generate a compiled program. The method comprises a step of (A) generating a first speculative instruction that is capable of experiencing an instruction exception condition during execution of the first speculative instruction, wherein the first speculative instruction defers signaling an instruction exception when the exception condition is initially detected and completes execution without signaling the instruction exception.

A further illustrative embodiment of the invention is directed to a computer system, comprising means for executing a first program instruction based upon a speculation, the first program instruction being capable of experiencing an instruction exception condition during execution of the first program instruction; means for deferring signaling of an instruction exception when the exception condition is detected during execution of the first program instruction; means for determining whether the speculation was incorrect; and means for ignoring the instruction exception when the speculation was incorrect.

Another illustrative embodiment of the invention is directed to a computer readable medium encoded with a program that, when executed on a computer system, performs a method including steps of: executing a first program instruction based upon a speculation, the first program instruction being capable of experiencing an instruction exception condition during execution of the first program instruction; deferring signaling of an instruction exception when the exception condition is initially detected during execution of the first program instruction; determining whether the speculation was incorrect; and ignoring the instruction exception when the speculation was incorrect.

Brief Description of the Drawings

Figure 1 is a block diagram of a general purpose computer on which embodiments of the present invention can be implemented;

Figure 2 depicts an original code sequence including three basic blocks;

Figure 3 depicts a scheduled code sequence resulting from scheduling the original code sequence of Figure 2 using static speculation according to one embodiment of the present invention;

Figure 4 depicts an original code sequence including a memory store instruction followed by a memory load instruction;

Figure 5 depicts a scheduled code sequence resulting from scheduling the original code sequence of Figure 4 using static data speculation to advance the load instruction according to one embodiment of the present invention;

Figure 6 is a flowchart of a process for advancing a load instruction and instructions dependent thereon ahead of a store instruction according to one embodiment of the present invention;

Figure 7 depicts a flowchart of a process for executing advanced load instructions at runtime according to one embodiment of the present invention; and

Figure 8 depicts an example of an advanced load address table according to one embodiment of the present invention.

Detailed Description

One embodiment of the invention is directed to a method and apparatus for allowing recovery from problems encountered during execution of an advanced or speculated instruction.

This aspect of the present invention can be employed with any type of computer system. An example of such a computer system is the general purpose computer 50 illustrated in Figure 1. The general purpose computer 50 includes a processor 52, an input device 54, an output device 56, and memory 58 connected together through a bus 60. The memory 58 includes primary memory 62 (i.e., fast volatile memory such as a dynamic semiconductor memory) and secondary memory 64 (i.e., nonvolatile memory such as magnetic disks). The memory 58 stores one or more programs 66 executed on the processor 52.

The programs 66, when executed by the processor 52, control the general purpose computer 50. The programs 66 may include a compiler, the function of which is described below in connection with Figure 6.

It should be appreciated that the computer system 50 of Fig. 1 is provided merely for illustrative purposes, and that the embodiments of the present invention described below can be implemented on computer systems of numerous other types and configurations. One aspect of the present invention provides a method and apparatus for recovering from problems encountered during execution of statically speculated instructions. One embodiment of the present invention is directed to executing any type of instruction segment that has been scheduled by a compiler to be speculatively executed, verifying the integrity of the execution of the instructions that were speculatively executed, and executing recovery code to correct problems, if any problems are detected.

Instructions are divided into two classes: speculative and non-speculative. At the start of compilation all instructions are initialized to be non-speculative. When, during the course of scheduling, the compiler schedules an instruction outside of the instruction's originating basic block, the compiler marks the instruction as being speculative. Non-speculative instructions that encounter an exceptional condition generate an exception. Speculative instructions that encounter an exceptional condition do not generate an exception but rather write a "deferred exception token" (DET) into their destination. The existence of the exceptional condition prevents a specified computation instruction from being completed with the proper operands and therefore the destination of the instruction includes the DET rather than the correct result. A non-speculative instruction that reads a DET generates an exception. A speculative instruction that reads a DET writes another DET into the instruction's destination (note again that the destination does not contain the correct result) - this behavior is called "propagation." By placing a non-speculative instruction into the originating basic block of a particular speculative instruction, and by configuring the non-speculative instruction to read a destination of the speculative instruction (or any location into which a DET may have been propagated), a DET generated by the speculative instruction can be detected at the point at which control is passed to the originating basic block. At this point it is necessary to re-create the exceptional condition that caused the original creation of the DET, and to replace all previously propagated DET's with the correct results. This is achieved by a process called "recovery." Recovery can involve augmenting the program with additional code generated by the compiler; the code is a copy of the set of dependent speculative instructions in non-speculative form such that, upon execution, all exceptional conditions generate exceptions and all previously written destinations are overwritten with the correct results. The recovery code need not be an exact copy of the instruction sequence, but may be code that, when executed, achieves the same result. Furthermore, in one embodiment of the invention, a new instruction is defined with the specific purpose of checking for the existence of DET's and activating the associated recovery code in the event a DET is detected.

The above-discussed embodiment of the present invention does not depend on the exact form of the DET. Also, alternative embodiments for specifying speculative and non-speculative instructions are possible without affecting the spirit or scope of the present invention. For example, some instructions may be defined to behave speculatively independent of whether the instructions were scheduled outside of their originating basic block.

The speculation referred to up to this point is called "control speculation" since instructions are executed before control is passed to them. Speculation can take other forms besides control speculation. One example of this is "data speculation" whereby a mechanism is defined that allows instruction A, which may be dependent on instruction B, to be executed before instruction B. Although data speculation can apply to any class of instruction, loads and stores are discussed below to demonstrate data speculation. A load that is below a store generally cannot be scheduled above the store unless it can be shown that the address read by the load is never equal to the address written by the store. If the addresses are equal then the load should receive the results of the store. However, if it can be shown that the address read by the load is never equal to the address written by the store, then the load can be safely scheduled above the store. Data speculation occurs when the compiler schedules the load above the store when it cannot be proven that the addresses being accessed by both will never be equal. When the addresses accessed by both instructions are determined to be equal at runtime, an error condition known as a collision occurs. In the event of a collision, a recovery mechanism may be employed to correct any incorrectly written destinations. In one embodiment of the present invention, a load instruction and one or more instructions dependent thereon are scheduled by the compiler above a store instruction, even if the compiler determines there may be a collision between the two instructions. Thus, aspects of the present invention are directed to control speculation, data speculation, as well as other forms of speculation.

One aspect of the present invention provides a technique wherein a compiler may schedule instructions to be speculatively executed, yet the computer system can recover from speculation errors that occur during speculative execution of the instructions. Another aspect of the present invention is directed to a method and apparatus for advancing an instruction out of order. This includes scheduling a second instruction, as well as an entire calculation dependent thereon, (e.g., a load instruction) to be executed in advance of a first instruction (e.g., a store instruction) which may produce an error condition known as a collision, due to the first and second instructions accessing the same address in a portion of a memory.

To implement some aspects of the present invention, a computer architecture can be defined that allows a compiler to schedule instructions outside of their originating basic block (control speculation), and schedule parallel execution of instructions that may potentially access the same memory location (data speculation), and therefore are potentially dependent. One example of such a computer architecture is described in greater detail in copending U.S. Patent Application Serial No. 08/949,295, filed on October 13, 1997, entitled "COMPUTER ARCHITECTURE FOR THE DEFERRAL OF EXCEPTIONS ON SPECULATIVE INSTRUCTIONS" by Jonathan K. Ross et. al., which is incorporated herein by reference. While the aspects of the present invention are described below with respect to this architecture, the present invention is not limited to use with this architecture, and may be realized using other architectural features, as will be described in greater detail below.

This new architecture defines a set of "speculative" instructions that do not immediately signal an exception when an exceptional condition occurs. Instead, a speculative instruction defers an exception by writing a "deferred exception token" (DET) into the destination specified by the instruction. The instruction set also contains "non-speculative" instructions that immediately signal an exception when an exceptional condition occurs, as is common of conventional instructions.

Instruction exceptions are well known in the art, and include, but are not limited to, exceptions such as page faults, illegal operands, privilege violations, divide-by-zero operations, overflows, and the like. The new architecture also provides a new type of memory speculation wherein a load instruction that follows a store instruction in the logical order defined by a programmer may be executed before the store instruction based on the speculation that the two instructions will not access the same memory location. A memory speculation check, which may, for example, access an advanced load address table (ALAT) that contains a record of recent speculative memory accesses, can be provided to determine if the speculation is correct. If the speculation is correct, the instructions were properly executed. If not, the load instruction, and any instructions that are dependent on the load and were scheduled above the store, are executed again to retrieve the contents written by the store instruction.

Using instructions marked as speculative, a compiler may schedule instructions outside of their originating basic block, and may schedule possibly dependent memory accesses in parallel. As stated above, when a speculative instruction generates an exception, a "deferred exception token" may be written into the destination specified by the instruction. Any speculative instruction that detects a DET in any source copies the DET into its destination. Note that when a speculative instruction finds a DET in a source, it need not perform the function associated with the instruction. The instruction can simply copy the DET to the destination. In this manner, the DET propagates through the block of speculative instructions. Thus, in one embodiment of the invention a destination which would include the result of a calculation may be checked for a DET, without checking each operand used in the calculation.

Any non-speculative instruction that detects a DET in a source may generate an immediate exception. Accordingly, DET's propagate through speculative instructions in dataflow fashion until (and if) they reach a non-speculative instruction.

If a program determines at runtime that the speculation upon which instructions were executed was incorrect (for example, a branch that was incorrectly predicted), the program may simply ignore the DET's since the DET's are not accessed by the program. However, if the speculation was correct, DET's are converted into an actual exception if, and when, the originating basic block of the instruction that created the DET is executed. In one embodiment, this conversion is performed by an instruction called the "speculation check" instruction, or "chk.s" for short. The chk.s instruction reads a source, and if the source contains a DET, branches to a specified target address that implements recovery code. Similarly, in one embodiment of the present invention the correctness of memory speculation may be determined by an "advance check" instruction, called a chk.a instruction. The chk.a instruction determines whether a memory location was accessed out-of-order, and if it was, the chk.a instruction branches to a specified target address that implements recovery code. The chk.a instruction will be discussed in greater detail below. Chk.s and chk.a may each be implemented in a number of ways which result in a change in the control flow of the instructions being executed. For example, each can be implemented as a conditional branch instruction, or as an instruction that generates an exception that invokes an exception handler.

By definition, the chk.s and chk.a instructions are always non-speculative. Generally, if these instructions detect a DET or an incorrect memory speculation, recovery code is executed that includes non-speculative versions of the offending instructions. With respect to a chk.s instruction that detects a DET, upon execution of the recovery code, the non-speculative version of the offending instruction will replace the DET in its destination with the correct result and/or generate the exception. If any later speculative instructions were dependent on the offending instruction, they are also included in the recovery code to be re-executed because the DET's were propagated into the later speculative instruction's destinations, and therefore these destinations may contain incorrect results. With respect to a chk.a instruction, the recovery code must re-execute the offending load instruction to load the proper contents from memory. In addition, any instructions dependent on the offending load instruction that were scheduled above the store on which the load depends are also re-executed. Scheduling of load instructions and calculation instructions dependent on the loaded value above the store instruction will be discussed further below. Any instructions not dependent on the offending instruction are not re-executed since they will incorrectly modify the program state. Since the compiler scheduled the speculative instructions and the speculation checks, the compiler will be able to generate recovery code appropriate for a particular set of speculative instructions.

One aspect of the present invention may be realized by a computer system having a compiler capable of scheduling instructions for speculative execution and generating appropriate recovery code, and an architecture capable of executing instructions marked as speculative, such as a computer system that implements the architecture described above.

Figure 2 depicts an original code sequence 10 consisting of three basic blocks A1, B1, and C1. Original code sequence 10 represents code as specified by a programmer. Within code 10, instruction I0 represents instructions coming before instruction I2. Instruction I2 is a branch instruction that branches to instruction I14 if the contents of register r0 are non-zero. Instruction

I4 loads register rl with the contents of the memory location pointed to by register r2. Instruction I6 shifts the contents of register rl left by three bit positions, and writes the result into register r3. Instruction I8 adds the contents of registers r3 and r5, and writes the result in register r4. Instruction I10 compares the contents of register r4 with the contents of register r7. If the contents of register r4 are greater than the contents of register r7, then a non-zero value is written into register r6. Otherwise, zero is written into register r6. Instruction I12 is a branch instruction that branches to instruction I100 (not shown in Figure 2) if the contents of register r6 are non-zero. Finally, instruction I14 represents instructions that come after instruction I12 when the branch is not taken. Within basic block B1, instruction I12 is dependent on instruction I10, which in turn is dependent on instruction I8, which in turn is dependent on instruction I6, which in turn is dependent on instruction I4.

Figure 3 depicts a scheduled code sequence 20 resulting from scheduling original code 10 of Figure 2 using static speculation in accordance with one illustrative embodiment of the present invention. In Figure 3, instructions I4, I6, and I8 have been scheduled outside of their originating basic block B1 and into block A1, and have therefore been marked as speculative by the compiler (indicated by the ".s" modifier). Instructions I10 and I12 have not been scheduled outside of their originating basic block B1 and are not marked with an ".s" since they are not speculative.

In one embodiment of the present invention, certain instructions, generally those that do not cause exceptions, always behave as if they were speculative (and, for example, propagate DET's) independent of whether or not they were scheduled outside of their originating basic block. Therefore, these instructions are not explicitly marked as speculative or non-speculative. Certain other instructions that cause exceptions, such as load instructions, may have both speculative and non-speculative varieties. Therefore, the compiler will explicitly mark these as speculative or non-speculative depending on how they are scheduled. The present invention also applies to alternative embodiments such as those where every instruction is explicitly and individually marked as speculative or non-speculative.

A sequence of dependent speculative instructions beginning with the earliest speculative instruction and ending with the latest speculative instruction, all from the same basic block, is called a "speculative dependence chain" (as used herein, "early" and "late" are defined by the original program order). In the code shown in Figures 2 and 3, the speculative dependence chain begins with instruction I4, includes instruction I6, and ends with instruction I8. If any instruction in the speculative dependence chain encounters an exceptional condition, then a DET will be written into the offending instruction's destination and will propagate down the speculative dependence chain. For example, if instruction I4 encounters an exceptional condition, such as a page fault, a DET will be written into register r1. Instruction I6, upon reading a DET from register rl, will in turn write a DET into register r3. Likewise, instruction I8, upon reading a DET in register r3, will in turn write a DET into register r4. Note that in this example, instruction I6 need not perform the shift function specified by the shl.s instruction, and instruction I8 need not perform the add function specified by the add.s operation. This instruction may simply propagate the deferred execution token. Accordingly, once a deferred execution token is generated, execution resources which would otherwise be consumed by the execution of speculative instructions may be made available to execute other instructions or may remain dormant thus reducing power dissipation.

At instruction I2, register r0 is evaluated. If register r0 is non-zero, execution branches to instruction I14, in which case the value stored in register r4 is not required since instructions I4, I6, and I8 were executed based on an incorrect speculation, and any exception generated by instructions I4, I6, or I8 may be ignored. Since the compiler knows that instruction I14 and the instructions that follow will only be executed if instructions I4, I6, and I8 should not have been executed, instruction I14 and the instructions that follow can simply ignore the results placed in registers rl, r3, and r4 and reuse these registers for other purposes. It is the responsibility of the compiler to generate code that properly addresses the effects of instructions that were speculatively executed because of an incorrect speculation.

However, if register r0 is zero, then the results of instructions I4, I6, and I8 are validated. During scheduling by the compiler, at the time the first instruction is made speculative from a particular basic block (instruction I4 in this example), a chk.s instruction (instruction I9 in Figure 3) is emitted by the compiler and placed in that basic block (BI in this example). As noted earlier, the chk.s is non-speculative and is not scheduled outside of the basic block into which it is placed. The chk.s at instruction I9 reads register r4, which is the destination register of instruction I8. Instruction I9 verifies the results of all instructions in the speculative dependence chain above, including the instructions whose destination is read by the instruction I9, which are instructions I4, I6, and I8.

If a DET was not generated by the execution of instructions I4, I6, and I8, then instructions I4, I6, and I8 have been validated, thereby confirming that the speculative execution of these instructions was successful. Accordingly, execution continues with instruction I10.

However, if instruction I4, I6, or I8 generated a DET, then that DET will have propagated to register r4, where instruction I9 will detect the DET. Instructions in the speculative dependence chain (instructions I4, I6, and I8) will have produced unreliable results because their destination registers will contain DET's in place of correct results. Accordingly, the chk.s instruction (I9) will detect the DET and will branch to recovery code starting at instruction I4r. Instructions I4r, I6r, and I8r are non-speculative versions of instructions I4, I6, and I8, respectively, and instruction I9r branches to instruction I9 to re-execute the chk.s instruction. While it may not always be necessary to re-execute instruction I9, there are many situations were it is good practice to do so, such as when speculative dependence chains are dependent on each other.

Since instructions I4r, I6r, and I8r are non-speculative, they will not defer exceptions. Therefore, exceptions will be generated and processed. For example, assume that instruction I4r generates a page fault. Control will pass to the exception handler responsible for addressing page faults, and the fault will be processed. For example, program execution may be halted, or a memory page may be read in from a virtual memory swap file.

As noted earlier, to preserve the correct program state, only instructions from the offending speculative dependence chain are allowed to modify the processor state during execution of the recovery code. In the example shown in Figure 3, only instructions I4, I6, and I8 are re-executed as instructions I4r, I6r, and I8r, and the other instructions are not. This selective re-execution is achieved by making a copy of all instructions in the speculative dependence chain starting with the earliest instruction and ending with the instruction whose destination is read by the chk.s instruction. This copy is called the "recovery code", and the chk.s instruction transfers control to the recovery code in the event that the chk.s instruction encounters a DET. At the end of the recovery code the compiler emits a branch back to instruction I9. Since the recovery code is only executed if the corresponding chk.s instruction is executed, and since the chk.s is always non-speculative, all instructions in the recovery code are non-speculative. Thus, the instructions in the recovery code are converted to non-speculative versions (if necessary). In the example shown in Figure 3, the mainline versions of instructions I4, I6, and I8 are all marked speculative, while the recovery code copies (instructions I4r, I6r, and I8r) are all marked non-speculative. The same recovery code can be targeted by multiple chk.s instructions. Furthermore, the same speculative dependence chain may have multiple recovery codes associated with it by having separate chk.s instructions branch to separate recovery code segments.

The existence of a DET indicates that an exceptional condition occurred on an instruction in a speculative dependence chain. Therefore, before any instructions are re-executed, the exceptional condition is first handled by activating the associated exception handler. One embodiment of the present invention meets this requirement automatically because the recovery code contains non-speculative copies of all relevant instructions in the speculative dependence chain, and non-speculative instructions immediately signal exceptions. Upon execution of the recovery code, the original exception will be re-generated by the offending instruction and the appropriate exception handler will be activated. After the exception handler corrects the exceptional condition, control is returned back to the recovery code, which continues executing the remainder of the instructions before returning to the mainline code.

The present invention is not dependent upon any particular DET format. In the preferred embodiment the DET simply indicates that a deferred exception exists and contains no further information. Alternative embodiments can define the DET to contain other information that may be needed by a particular exception handler, e.g. the type of exception, the address of the offending instruction, etc.

Other aspects of the present invention also provide for recovery from other types of speculation, such as data speculation. In one embodiment of the present invention, load instructions that are advanced out of turn ahead of store instructions are used to illustrate data speculation and are described in reference to Figures 4-6. As used herein, the references to load and store instructions respectively indicate any instructions that perform reads and writes to memory, irrespective of whether the instructions perform other functions. Load instructions typically require longer amounts of time to execute than other instructions, due to memory latency. By moving a load instruction earlier in the execution of a program, efficiency of executing instructions in a computer is improved. The load, referred to as an advanced load, allows an increase in the parallelism of activities being performed that require use of the memory.

As discussed briefly above, often a compiler cannot detect with one hundred percent certainty whether a load instruction and a store instruction will collide (i.e., access a common memory location). This presents a barrier to achieving parallelism in that it forces a more conservative instruction schedule that will not overlap the load latency, i.e., will not move a load ahead of a store with which it might collide. However, in a large percentage of these cases, the load and store instructions do not in fact collide. Thus, one embodiment of the present invention allows a load instruction and calculations dependent thereon to be executed before a potentially colliding store instruction as one means of improving parallelism of program execution in a single or multiple processor system.

Consider the simple original code 30 shown in Figure 4. Code 30 includes: instruction I22, which stores the contents of register r3 into a memory location indexed by the contents of register rl; instruction I24, which loads register r4 with the contents of the memory location indexed by the contents of register r2; and instruction I26 which adds registers r4 and r6 and writes the result into register r5. Assume that when the compiler schedules code 30, it determines that it is unlikely, but not impossible, that the contents of register rl will be the same as the contents of register r2 when instructions I22 and I24 are executed. Further, assume that the compiler determines that it would be more efficient to schedule instructions I24 and I26 before (or in parallel with) instruction I22. With respect to scheduling instructions in parallel, it should be appreciated that even in a single processor system, the single processor will typically include multiple execution units on which multiple instructions can be executed in parallel.

Figure 5 shows scheduled code 40, which was produced when the compiler scheduled original code 30 of Figure 4, in accordance with one embodiment of the present invention. Code 40 includes instructions I24 and I26 scheduled before instruction I22 (the store instruction). Note that ".a" (indicating an advanced instruction) has been appended to the load instruction, indicating that this load instruction records the load address in the advanced load address table (ALAT). Instruction I25 is a chk.a instruction that checks the ALAT to determine whether the load (I24) and store (I22) instructions accessed the same memory location. If the contents of registers rl and r2 were not equal, the instructions did not access the same memory location, and chk.a (I25) takes no action. However, if the contents of registers rl and r2 were equal, the chk.a instruction (I25) detects a data speculation error and branches to the recovery code starting at instruction I24r. Instruction I24r re-executes the load instruction, causing the proper results to be loaded into register r4 since the load instruction is re-executed after the store instruction (I22). Instruction I26r re-executes the add instruction which writes the correct result into register r5 and instruction I23r branches back to instruction I25, which verifies that there is no data speculation error.

Figure 6 is a flowchart illustrating one exemplary routine that may be implemented by a compiler to generate the scheduled code changes shown in Figures 3 and 5. This flowchart is provided merely as an example, as other implementations are possible. The present invention is not limited to use with a specific program language or computer configuration, and may be applied to a compiler used in a single or multiple processor system.

The process of Figure 6 begins in step 503 when the compiler creates a conventional dependency graph representing instructions in a source computer program which has not yet been scheduled by the compiler. The present invention is not limited to any particular type of graph. The dependency graph may take several forms, including a diagram with at least one path representing a segment of the source computer program and a node representing each instruction in the segment of the computer program. The dependency graph for each program typically includes a plurality of paths, each with a plurality of nodes. A node representing an instruction may be annotated with information related to the instruction, such as the number of clock cycles needed to execute the instruction. Typically, arcs connecting nodes in the diagram indicate a dependency between instructions.

In step 506, the compiler reviews the graph and determines which path of the graph includes a sequence of instructions that results in a longest total execution time from start to finish.

In step 509, the compiler attempts to optimize the execution of the longest path in the program, since the longest path represents the critical portion in the program that limits the runtime of the instruction sequence. Conventional optimization techniques may be used by the compiler, and further optimization techniques are described below as they relate to an advanced load and its dependent calculations.

As mentioned above, one way the compiler can optimize the longest critical path is through data speculation. In one embodiment of the present invention, this may include moving an instruction, such as a load instruction which includes a read operation, earlier in the execution of the program before a store instruction which includes a write operation. In step 512, the compiler determines if a load instruction is in the longest critical path. If a load is included in the longest path, the compiler may advance the load and instructions dependent thereon as one method of optimization, as discussed further below.

When a load instruction is found in a path to be shortened, the compiler next determines, in step 521, which calculation instructions are dependent on the data read by the load instruction. The calculations are dependent if they require the use of the value resulting from the completion of the load instruction.

In step 524, the load instruction is removed from its place in the scheduled instruction sequence. In step 527, the calculations dependent on the load (identified in step 521) are advanced to follow the load instruction, such that both the load instruction and calculations dependent on the load instruction are advanced to allow optimization of the instruction sequence. The compiler advances the load instruction labeled "ld.a" ahead to a location wherein its execution may result in improved overall performance of the program.

As discussed above, a store instruction may exist in a path of the dependency graph ahead of the load instruction "ld.a", and the compiler may be unable to determine whether the load and store will collide (i.e., use the same memory location). In step 530, the compiler determines whether there is absolute certainty that the load and the store will not collide.

When it cannot be determined for certain that the load and the store in front of which it is moved will not collide, then in step 533, the load instruction which was removed in step 524 is replaced with a check instruction, such as a chk.a instruction described above in connection with Figures 3 and 4. The chk.a instruction replaces the load instruction and is performed (as described below) where the load would have been scheduled if it were not advanced.

In step 536, the compiler generates recovery code for the advanced load and the calculations dependent on the advanced load that were advanced in step 527. The recovery code will be called by the chk.a instruction, if necessary, as described below.

When a load instruction is not found in the longest critical path in step 512, or when it is determined at step 530 that the compiler is absolutely certain that the load has not been advanced in front of a store instruction with which it will collide (so that no check instruction or recovery code are necessary), or after the recovery code is generated in step 536, then the process continues at step 539, wherein the compiler determines if there is a longest critical path remaining which may potentially be optimized. The compiler may optimize each of the next longest paths in the program until the compiler optimizes as much of the source program as possible, thereby improving the parallelism of the execution of the source program.

When it is determined at step 539 that the compiler cannot further optimize the program, the process proceeds to step 542, wherein the compiler schedules the optimized instruction sequences for execution. However, when the compiler determines in step 539 that there is a longest critical path remaining which may potentially be optimized, the compiler identifies the next longest path, in step 506. In this manner, the process continues until all of the paths in the graph that can be optimized are optimized.

In step 542, the compiler schedules execution of the instructions to reflect any changes in execution order as carried out by the optimization procedures described above. The compiler may schedule execution of the instructions in the program in a number of ways, potentially making use of parallel execution units, and the present invention is not limited to any particular scheduling mechanism. For the example described above in Figure 4, the resulting optimized code is shown in Figure 5 and the recovery code is noted as instructions I24r, I26r and I23r.

Figure 7 is a flowchart illustrating a routine executed by a computer system when executing an instruction sequence optimized through techniques, such as those described above in connection with Figure 6, that include advancing a load instruction and dependent calculations out of order ahead of a store.

Execution of the scheduled instruction sequence is started in step 603. During the execution of the instruction sequence, the advanced load (e.g., ld.a, in Figure 5) is executed in step 606. After the advanced load instruction has been executed, the ALAT is updated to record the range of memory locations read by the advanced load instruction (e.g., the address in r2) executed in step 609. An entry is made in the ALAT to allow a later executed store instruction, such as the store executed in step 618, to determine if the advanced load and the store instructions accessed a common memory location. The present invention is not limited to any particular ALAT structure, as any structure that allows the range of memory addresses of a particular advanced load and its corresponding store instruction to be compared may be used.

In one embodiment shown in Figure 8, an entry in the ALAT includes a physical memory address field and a memory access size field that together define the range of memory locations accessed. The present invention is not limited to this method of defining the range of memory locations, as numerous other techniques can be employed. For example, the memory range accessed may be identified by start and end memory addresses, or by a memory end address and a range. In the embodiment shown in Fig. 8, the ALAT also includes a field for a valid bit that is used to indicate whether the entry is valid. As discussed below, in one embodiment of the invention, there are times (e.g., context switches between two applications) when it is desirable to invalidate the entries in the ALAT. The valid bit provides a convenient mechanism for performing such invalidation.

In one embodiment of the invention, there can simultaneously be multiple loads that each has been advanced ahead of a corresponding store. As discussed below, during program execution a technique is provided for verifying that no collision occurred between each pair of loads and stores. Thus, in one embodiment of the invention, the ALAT includes information that uniquely identifies the advanced load instruction to which it corresponds, so that the entry can be identified to determine a possible collision with the corresponding store instruction. This unique identification can be implemented in any number of ways, and the present invention is not limited to any particular implementation. In the embodiment shown in Figure 8, the entry for a particular advanced load is indexed based on the register number and type of register (general or floating point) used in the advanced load instruction. The register number used for each instruction is assigned by the compiler, which ensures that a unique register is used for each advanced load instruction.

When the advanced load is executed in step 606 (Figure 7) and before an entry in the ALAT is made, the ALAT is accessed using the target register number as an index and the ALAT is checked to determine whether an entry already exists corresponding to the target register number used by the advanced load. If there is an entry, it is removed because it includes information that is not related to the present advanced load and was most likely entered during execution of an earlier advanced load. After the existing entry is cleared of the data from the previously executed advanced load, or when an empty slot corresponding to the target register number of the present advanced load is found in the ALAT, a new entry indexed by the target register number is made for the present advanced load instruction.

After the advanced load is executed and the ALAT is updated, the calculation instructions dependent on the result of the advanced load instruction (e.g., I24, I26) are executed in step 612. In the example discussed above, the dependent calculations include an add instruction I26, as shown in Figure 5. Any other instructions that follow the advanced load instruction and that precede the store instruction above which the load was advanced are executed in step 615.

In step 618, the store instruction (e.g., I22) above which the load was advanced and with which the load may possibly collide is executed. In the example shown in Figure 5, the store instruction I24 accesses the address in r1. In executing the store instruction I22, all of the valid entries in the ALAT are searched using the physical address and size of the region of memory being written by the store. This searching can be done in any number of ways, and the present invention is not limited to any particular technique. In one embodiment of the present invention, the ALAT is arranged as a fully associative content addressable memory so that all of the valid entr·-s are searched simultaneously. The entries are searched to determine if a collision occurred bet· een the store instruction and any advanced load instruction. If the range of memory space of an advanced load (e.g., I24) is found in the ALAT that overlaps the range of memory space for the store instruction, (e.g., I22), then a collision has occurred. In one embodiment of the invention, when a collision is detected, the entries in the ALAT for the addresses corresponding to the colliding advanced loads are removed in step 621, signifying the collision. In step 621, if the memory space accessed by the store instruction (e.g., I22) does not overlap with that of any advanced load instruction (e.g., I24), the entries in the ALAT corresponding to the particular advanced load instructions remain in the ALAT.

It should be appreciated that a load instruction which is advanced may be moved ahead of multiple store instructions, each of which may potentially collide with the load instruction. A single check instruction following a sequence of store instructions may be used to detect if any of the multiple store instructions collide with the load instruction since executing each store instruction includes searching all of the entries in the ALAT to determine if a collision occurred. Thus, the check instruction is independent of the number of store instructions in the program since a separate check instruction replaces each load instruction which is advanced in step 533 and each check instruction reviews the ALAT as described below in step 623.

It should also be appreciated that a collision can occur between an advanced load and a store, as discussed above, even if the starting addresses for the data accessed by those instructions are not identical. In particular, each instruction may access multiple bytes of data. Thus, a collision can occur if there is any overlap between the range of memory addresses occupied by the data read by the advanced load, and the range of memory addresses occupied by the data written by the store. The detection of a collision can be performed in numerous ways, and the present invention is not limited to any particular implementation. For example, a full range comparison can be performed between the addresses for the data written by the advanced load and read by the store. However, a full range comparison can be expensive to implement in hardware. Therefore, in accordance with one embodiment of the invention, a technique is employed for determining collisions without performing a full range comparison of the addresses for the data accessed by the advanced load and the store.

In accordance with this embodiment of the present invention, a preference is given for size-aligning data stored in memory, so that the starting address for a block of data is preferably an even multiple of its size. For example, a block of data including four bytes is preferably stored at an address wherein the two least significant bits (LSBs) are zeros, a block of eight bytes is preferably stored at an address wherein the three LSBs are zeros, etc. When the data is size-aligned, a collision can be detected by simply performing a direct equality comparison of the starting addresses for the data accessed by the advanced load and the store. It should be appreciated that a direct equality comparison is significantly less costly to implement in hardware than a full range comparison. When data is misaligned, if there is a limited amount of misalignment such that the misaligned data can fit in a larger, size-aligned data range, then in one embodiment of the invention, the hardware processes the instruction, (e.g., a load) which accesses the misaligned data as if it were accessing the larger size-aligned data range. For example, if a load accesses eight bytes of misaligned data in memory, but the data accessed by the load would fit into a size-aligned thirty-two byte range, then the load is treated as accessing thirty-two bytes of data. It should be appreciated that by treating the instruction as employing a larger block of size-aligned data, there may be situations where there is an overlap of the data addresses accessed by an advanced load and a store instruction, but not an overlap of the actual data (eight bytes in the example above) actually accessed by one of the instructions, resulting in the detection of a false collision. This performance penalty is a price paid to reduce the complexity of the hardware for detecting collisions. If data is significantly misaligned and does not fit into a reasonably large size-aligned data range, then the hardware will not process the load or store instruction. For loads, no entry is inserted into the ALAT, which causes a collision to be indicated. Although this may result in false collisions being detected, this performance penalty is a price paid to reduce the complexity of the hardware for detecting collisions. For store instructions which are significantly misaligned, the instruction may be separated into a sequence of smaller stores instructions. In one embodiment, hardware may be used to separate the larger store into the sequence of multiple stores. In another embodiment, a misaligned store which cannot fit into a size-aligned data range may cause an interrupt, and the operating system handles the store by separating the store instruction into a sequence of smaller store instructions. In both embodiments of handling a misaligned store instruction, each of the smaller stores in the sequence is checked against the valid entries in the ALAT, as described in step 618. If the range of memory space for any of the smaller stores overlaps the range of memory space of an advanced load, then a collision would be indicated as described in step 621. Thus, executing each of the smaller stores provides the same result as executing a single larger store.

In another embodiment of the present invention, a further savings in the hardware employed to detect collisions is achieved by using partial addresses for the equality comparison by ignoring one or more of the most significant bits (MSBs) of the addresses. Ignoring one or more of the MSBs results in a reduction in the size of the ALAT and in the hardware that performs the equality comparison, since fewer bits are stored in the ALAT for each entry and fewer bits are compared. For example, for a 64-bit data address, only the twenty least significant bits (LSB) of the load can be saved in the ALAT and used in the equality comparison.

It should be appreciated that ignoring one or more of the MSBs may result in the detection of some false collisions. In particular, when the ALAT is searched in executing a store instruction (e.g., in step 618), the complete starting addresses for the data of the store and the data of the load may be identical for the LSBs over which the equality comparison is performed (e.g., the twenty LSBs), but may differ for one or more of the MSBs that are ignored. When this occurs, the routine of Fig. 7 will perform as if a collision had actually occurred, e.g., by switching control flow to the recovery code. It should be appreciated that false detections will therefore result in some performance penalty due to recovering from collisions that did not in fact take place. This performance penalty is a price paid to reduce the complexity of the hardware for detecting collisions. A balance between these competing factors can be considered when determining how many (if any) MSBs to ignore in the detection scheme.

In step 623, the chk.a instruction (e.g., I25) is executed for the advanced load instruction executed in step 606. In one embodiment, the chk.a instruction reviews the ALAT to determine if a collision occurred by determining if there is an entry for the advanced load instruction (e.g., I24). Using as an index the identity of the target register and register type used by the advanced load instruction (e.g., I24) for which information was updated in the ALAT in step 621, the chk.a instruction reviews the ALAT. In step 624, if an entry is found in the ALAT corresponding to the particular advanced load replaced by the chk.a instruction, then the chk.a instruction recognizes that the store instruction (e.g., I22) and the load instruction (e.g., I24) advanced above the store did not collide. Therefore, the data read by the store instruction is valid, and the routine proceeds to finish execution of the instruction sequence in step 630.

However, if the chk.a instruction reviews the ALAT in step 624 and does not see an entry in the ALAT corresponding to the register address used by the advanced load instruction (e.g., I24), then the chk.a instruction (e.g., I25) determines that the store and advanced load instructions may have accessed the same memory space (i.e., collided). Thus, further steps are taken to ensure the accuracy of the advanced load instruction and the calculation instructions that have been executed based on the advanced load instruction (e.g., I24). In one embodiment, when a possible collision is detected, control flow of the program is changed to execute recovery code. As discussed above, this can be done numerous ways (e.g., by branching or using an exception handling technique). It should be appreciated that the ALAT can be implemented with a number of entries that may not be sufficient to support all of the advanced local instructions that may be included in a program being executed. In the embodiment shown in Fig. 7, the chk.a instruction branches to recovery code in step 633 (Figure 7). An example of recovery code is shown in Figure 5 as instructions I24r, I26r and I23r, which are essentially copies of comments I24, I26 and I23.

In step 636, the load instruction (e.g., I24) that was advanced in step 524 is re-executed. In step 639, instructions dependent on the advanced load instruction (e.g., I24) are re-executed. In the example of Figure 5, the re-executed load instruction I24r and dependent add instruction I26r are shown. These instructions are re-executed after the store instruction to provide the proper results of the load instruction I24r and its dependent calculations I26r. In one embodiment, the recovery code may be any combination of instructions determined by the compiler that provides the same result as the originally executed load and dependent calculation instructions.

In step 642, the control flow returns from the recovery code back to the compiled execution schedule. This can be done, for example, using a branch instruction such as I23r in Figure 5. Next, the execution of the scheduled instructions continues until the end of the program in step 630.

For example, in one embodiment the ALAT has space for entries corresponding to thirty-two advanced instructions. If there are more than thirty-two advanced instructions in an executed program, then the ALAT will not have enough space for information pertaining to all of the advanced instructions. When the ALAT is full and a new advanced instruction is executed, a replacement scheme can be used to take out a valid entry in the ALAT to make room for the new instruction. Furthermore, in one embodiment of the present invention, when execution at runtime switches between processes (such as between separate compiled programs), the entries in the ALAT may be saved for later restoration, or may be invalidated. Thus, in some instances, a chk.a instruction may not find an entry for a particular advanced instruction, even though a collision did not occur for that instruction.

As discussed above, the present invention is not limited to a particular ALAT structure and may include other alternative embodiments for determining whether there was a collision between a load and a store instruction. For example, other data structures or compare circuits may be used. Also, an ALAT or other structure used may vary in size and in the number of fields used. In addition, separate ALAT's or data structures may be used for each of multiple register sets. For example, in one embodiment an ALAT may be used for general and floating point register sets.

While the present invention has been explained with reference to deferred exception handling and data speculation, it is not so limited. In general, the present invention encompasses any type of instruction segment that is speculatively executed, verifying the integrity of the execution of the instructions that were speculatively executed, and executing recovery code to correct any problems detected. The present invention may be extended to include an instruction that is both control and data speculative.

The transfer of control from the chk.s and chk.a instructions to the recovery code can be implemented in any of a number of ways. For example, the chk.s and chk.a instructions each may behave as a branch instruction where the address of the first instruction in the recovery code is contained in the chk.s or chk.a instruction itself (as shown in Figure 3). Alternatively, the chk.s or chk.a instruction may generate a particular exception and the exception handler may use a value in the chk.s or chk.a instruction to identify the corresponding recovery code and then transfer control to that recovery code. The exception handler may also use the address of the chk.a or chk.s instruction, which is the address location in memory where the instruction is stored, to identify the location of the recovery code. The recovery code can be based on a table created by the compiler which includes addresses of check instructions which were added by the compiler to a compiled source program. The recovery code executed is therefore identified by which check instruction is executed.

The present invention allows instructions to be advanced out of turn, even when the compiler is not certain that the instruction advanced will not collide with a later instruction. As discussed above, some conventional compilers may advance a single load instruction moved ahead of a store even if not certain that the load and store will not collide. At runtime, if there was a collision, the load instruction would be re-executed in-line in the compiled execution schedule. In contrast, in one embodiment of the present invention, optimization of instruction execution is achieved by advancing not just a load ahead of store, but also calculations dependent thereon. This enables a compiler and scheduler to most efficiently use multiple execution units at one time. Further, instead of simply re-executing the load instruction if there is a collision, a check instruction is executed which determines if there was a collision and the control flow is changed to recovery code that includes the load instruction and its dependent calculations. Thus, multiple sections of code may be executed independently and in parallel.

The present invention allows flexibility on the part of the compiler regarding the association between chk instructions, the speculative dependence chain, and the recovery code. The example contained herein is relatively simple, but much more complex code configurations are possible, such as when a speculative dependence chain consists not of a single linear sequence of instructions, as in the example shown in Figure 5, but contains multiple sequences, or when two or more speculative dependence chains depend on each other. The present invention allows significant flexibility in addressing these various configurations, thereby allowing future refinements in the usage of recovery code as understanding of static speculation increases.

The present invention allows a wide degree of flexibility with regard to the number and configuration of chk instructions. For example, a single chk.s may be configured to read the destination of any one of the instructions along the speculative dependence chain, or multiple chk.s instructions may be emitted, with each reading a different destination. Each chk.s instruction may also invoke the same or a different set of recovery code instructions.

The present invention also encompasses alternative embodiments for detecting the presence of DET's. For example, in one embodiment, there is not an explicit chk.s instruction. Rather, DET's are detected by every non-speculative instruction as part of the normal execution of non-speculative instructions. In this embodiment, when a non-speculative instruction encounters a DET, an exception is generated that addresses the deferred exception. In another illustrative embodiment, DET's are stored in dedicated registers or memory, rather than the destinations of each instruction.

In another embodiment, the non-speculative recovery code may be the same code as the speculative in-line code. For example, an architecture may be employed wherein every instruction may be marked speculative or non-speculative based on a speculation flag contained in the instruction. For example, a compiler may schedule a segment of instructions as speculative, and a DET may be detected after these instructions have been executed, thereby activating a deferred exception handler. The deferred exception handler can simply toggle the speculation flag of the speculative instructions to convert them into non-speculative instructions, re-execute the instructions, process the exception that was previously deferred, and toggle the flags back to convert the instructions back to speculative instructions. While this embodiment provides the compiler with less flexibility in scheduling recovery code, it can also result in a substantial savings in the amount of memory consumed by the code. In addition, the speculation flag need only be toggled in cache memory, thereby minimizing the time required to toggle the speculation flags.

In a similar embodiment, a set of registers may be defined to identify code segments wherein speculative instructions should be executed as non-speculative. This embodiment would function substantially as described above, except that instead of toggling the speculation flags of the instructions to be executed non-speculatively, the registers would be loaded with indexes that identify the instructions to be executed non-speculatively.

It should be understood that various changes and modification of the embodiments shown in the drawings and described in the specification may be made within the scope of the invention. It is intended that all matter contained in the above-description and shown in the accompanying drawings be interpreted in an illustrative and not in a limiting sense. The invention is only limited as defined in the following claims and their equivalence thereto.

CLAIMS

1. A computer readable medium having a compiled program stored thereon in a computer-readable form, the program comprising:

a store instruction;

a load instruction that is scheduled before the store instruction;

at least one calculation instruction that is dependent on data read by the load instruction, the at least one calculation instruction being scheduled ahead of the store instruction; and

a check instruction to determine whether the store instruction and the load instruction access a common location in memory.

2. The computer readable medium of claim 1, wherein the check instruction is scheduled after the store instruction.

3. The computer readable medium of claim 1, wherein the store instruction includes a plurality of store instructions.

4. The computer readable medium of claim 1, wherein the check instruction changes control flow when the store instruction and the load instruction access a common memory location.

5. A computer readable medium of claim 2, wherein the check instruction accesses an address table to determine whether the store instruction and the load instruction access a common location in memory, wherein after the execution of the store instruction an absence of an entry corresponding to the load instruction in the address table indicates that a common location in memory was accessed by the store and load instructions

6. The computer readable medium of claim 4, wherein the program further includes recovery code to which control is passed when the store instruction and the load instruction access a common memory location, the recovery code including a re-execution of the load instruction and the at least one calculation instruction.

7. A computer system, comprising:

a memory;

means for executing a store instruction;

means for executing a load instruction before the store instruction;

means for executing, before the store instruction, at least one calculation instruction dependent on data read by the load instruction; and

means for determining whether the store instruction and the load instruction accessed a common location in the memory.

8. The computer system of claim 7, further including means for changing control flow to recovery code when it is determined that the store instruction and the load instruction accessed a common location in the memory.

9. The computer system of claim 7, wherein the store instruction includes a plurality of store instructions.

10. The computer system of claim 7, wherein the means for determining includes a table that includes information indicating whether the load instruction and the store instruction accessed a common location in the memory.

11. The computer system of claim 8, wherein the recovery code includes a copy of the load instruction and the at least one calculation instruction.

12. The computer system of claim 8, wherein the means for determining includes a table that includes information indicating whether the load instruction and the store instruction accessed a common location in the memory.

13. A computer system, comprising:

a compiler to create an execution schedule for a source program that includes a store instruction, a load instruction, and at least one calculation instruction that is dependent on data read by the load instruction, wherein the compiler includes means for scheduling the load and the at least one calculation instructions ahead of the store instruction when the compiler cannot be certain that the store and load instructions will not access a common memory location during execution of the program.

14. The computer system of claim 13, wherein the compiler includes means for generating a check instruction that, during execution of the program, determines whether the store instruction and the load instruction access a common memory location.

15. The computer system of claim 14, wherein the scheduler includes means for scheduling the check instruction after the store instruction.

16. The computer system of claim 14, wherein the compiler includes means for generating a check instruction that includes a change in control flow when the check instruction determines that the store instruction and the load instruction accessed a common memory location.

17. The computer system of claim 14, further including a plurality of execution units, and wherein the compiler further includes means for scheduling execution of the load and store instructions on different execution units.

18. The computer system of claim 16, wherein the compiler further includes means for generating recovery code to which control is passed when the store instruction and the load instruction accessed a common memory location, the recovery code including a copy of the load instruction and the at least one calculation instruction.

19. A computer readable medium having a compiled program stored thereon in a computer-readable form, the program comprising:

a store instruction;

a load instruction that is scheduled before the store instruction; and

a check instruction to determine whether the store instruction and the load instruction access a common memory location during execution of the program, the check instruction changing control flow when the check instruction determines that the store instruction and the load instruction accessed a common memory location.

20. The computer readable medium of claim 19, wherein the program further includes recovery code to which control is passed when the check instruction determines that the store

instruction and the load instruction accessed a common memory location during execution of the program, the recovery code including a copy of the load instruction.

21. A computer system, comprising:

a memory;

means for checking whether a store instruction and a previously-executed load instruction accessed a common location in the memory; and

means for changing control flow to recovery code when it is determined that the store instruction and the load instruction accessed a common location in the memory.

22. The computer system of claim 21, wherein the computer system includes means for generating the recovery code to include a copy of the load instruction.

23. The computer system of claim 21, wherein the means for checking includes a table that includes information indicating whether the load instruction and the store instruction accessed a common location in the memory.

24. A computer readable medium encoded with a compiler that, when executed on a computer system. performs a method of compiling a source program that includes a store instruction, a load instruction, and at least one calculation instruction that is dependent on data read by the load instruction, the method comprising steps of:

(A) determining whether the store and load instructions will not access a common memory location during execution of the program; and

(B) when it cannot be determined that the store and load instructions will not access a common memory location, scheduling the load instruction and the at least one calculation instruction ahead of the store instruction.

25. The computer readable medium of claim 24, wherein the method further includes a step of:

(C) generating a check instruction to determine whether the store instruction and the load instruction access a common memory location during execution of the program.

26. The computer readable medium of claim 25, wherein the step (C) includes a step of scheduling the check instruction after the store instruction.

27. The computer readable medium of claim 25, wherein the step (C) includes a step of generating a check instruction that changes control flow when the check instruction determines that the store instruction and the load instruction accessed a common memory location.

28. The computer readable medium of claim 27, wherein the method further includes a step of:

(D) generating recovery code to which control is passed when the check instruction determines that the store instruction and the load instruction accessed a common memory location during execution of the program, the recovery code including a copy of the load instruction and the at least one calculation instruction.

29. A computer readable medium encoded with a compiler that, when executed on a computer system, performs a method of compiling a program that includes a first instruction and a second instruction, wherein the compiler cannot be certain that the second instruction will not operate upon data that is dependent upon the execution of the first instruction, the method comprising steps of:

(A) scheduling the second instruction ahead of the first instruction; and

(B) generating a check instruction to, during execution of the program, determine whether the second instruction operates on data that is dependent upon the execution of the first instruction.

30. The computer readable medium of claim 29, wherein the step (B) includes a step of generating a check instruction that includes changing control flow when the check instruction determines that the second instruction operates upon data that is dependent upon the execution of the first instruction.

31. The computer readable medium of claim 30, wherein the method further includes a step of:

(C) generating recovery code to which control is passed, the recovery code including a copy of the second instruction.

32. A computer readable medium encoded with a compiler that, when executed on a computer system, performs a method of compiling a source program that includes a store instruction, a load instruction and at least one calculation instruction that is dependent on data read by the load instruction, the method comprising steps of:

(A) scheduling the load instruction and the at least one calculation instruction ahead of the store instruction;

(B) generating a branch instruction that branches when it is determined, during execution of the program, that the store and load instructions accessed a common memory location; and

(C) generating recovery code to which the branch instruction branches when it is determined that the store instruction and the load instruction accessed a common memory location during execution of the program, the recovery code including a copy of the load instruction and the at least one calculation instruction.

33. The computer readable medium of claim 32. wherein the step (B) includes a step of generating a check instruction that branches and determines whether the store instruction and the load instruction accessed a common memory location during execution of the program.

34. A method of executing instructions, comprising:

executing at least one instruction that has been marked as speculative;

verifying integrity of execution of the at least one instruction;

when the integrity of execution of the at least one instruction is verified, continuing executing other instructions; and

when the integrity of execution of the at least one instruction is not verified,

executing recovery code; and

continuing executing the other instructions after executing the recovery code.

35. A computer readable medium encoded with a compiler that, when executed on a computer system, performs a method of compiling a source program to generate a compiled

program that includes a plurality of instructions organized in a plurality of basic blocks, each basic block including a set of contiguous instructions, the plurality of instructions including a first instruction that is associated with a first basic block and that may generate an exception during execution of the compiled program, the method comprising steps of:

(A) scheduling the first instruction outside of the first basic block and ahead of at least one instruction that precedes the first basic block; and

(B) generating a check instruction to, during execution of the compiled program, determine whether the first instruction generates an exception, wherein the check instruction changes control flow when the check instruction determines that the first instruction generated an exception.

36. The computer readable medium of claim 35, wherein the method further includes a step of scheduling the check instruction within the first basic block.

37. The computer readable medium of claim 35, wherein the method further includes a step of identifying each of the plurality of instructions as speculative or non-speculative.

38. The computer readable medium of claim 35, wherein the method further includes a step of generating recovery code to which control is passed when the check instruction determines that the first instruction generated an exception, the recovery code including a copy of the first instruction.

39. A computer readable medium having a program stored thereon in a computer-readable form, the program comprising:

a plurality of instructions organized in a plurality of basic blocks, each basic block including a set of contiguous instructions, the plurality of instructions including;

a first instruction that is associated with a first basic block and that may generate an exception during execution of the program, wherein the first instruction is scheduled outside of the first basic block and ahead of at least one instruction that precedes the first basic block; and

a check instruction to, during execution of the program, determine whether the first instruction generates an exception, wherein the check instruction changes control flow when the check instruction determines that the first instruction generated an exception.

40. The computer readable medium of claim 39, wherein the check instruction is scheduled within the first basic block.

41. The computer readable medium of claim 39, wherein each of the plurality of instructions is identified as speculative or non-speculative.

42. The computer readable medium of claim 41, wherein the plurality of instructions further includes recovery code to which control is passed when the check instruction determines that the first instruction generated an exception.

43. The computer readable medium of claim 42, wherein the recovery code includes a copy of the first instruction.

44. A computer system, comprising:

means for executing a program including a plurality of instructions organized in a plurality of basic blocks, each basic block including a set of contiguous instructions, the plurality of instructions including a first instruction that is associated with a first basic block and that may generate an exception during execution of the program;

means for executing the first instruction outside of the first basic block and ahead of at least one instruction that precedes the first basic block;

means for determining whether the first instruction generates an exception; and

means for changing control flow to recovery code when it is determined that the first instruction generated an exception.

45. The computer system of claim 44, wherein the recovery code includes a copy of the first instruction.

46. A computer system, comprising:

a compiler to create an execution schedule for a program that includes a plurality of instructions organized in a plurality of basic blocks, each basic block including a set of contiguous instructions, the plurality of instructions including a first instruction that is associated with a first basic block and that may generate an exception during execution of the program, wherein the compiler includes means for scheduling the first instruction outside of the first basic block and ahead of at least one instruction that precedes the first basic block when the compiler cannot be certain that the first instruction will not generate an exception during execution of the program, and wherein the compiler includes means for generating a check instruction that includes a change in control flow when the check instruction determines that the first instruction generated an exception.

47. The computer system of claim 46, wherein the compiler further includes means for generating recovery code to which control is passed when the first instruction generates an exception, the recovery code including a copy of the first instruction.

48. A computer readable medium having a program stored thereon in a computer-readable form, the program comprising:

a first speculative instruction that is capable of experiencing an instruction exception condition during execution of the first speculative instruction. wherein the first speculative instruction defers signaling an instruction exception when the exception condition is initially detected and completes execution without signaling the instruction exception.

49. The computer readable medium of claim 48, wherein the first speculative instruction has a first destination, and wherein when the exception condition is initially detected, the first speculative instruction stores information in the first destination indicating that the instruction exception was detected during execution of the first speculative instruction.

50. The computer readable medium of claim 49, wherein the program further includes a second speculative instruction that is scheduled after the first speculative instruction and operates upon a result in the first destination, wherein the second speculative instruction has a second destination, and wherein when the information indicating that the instruction exception was detected during execution of the first speculative instruction is stored in the first destination, the

second speculative instruction stores information in the second destination indicating that the instruction exception was detected during execution of the program.

51. The computer readable medium of claim 50, wherein the program further includes a check instruction that is scheduled after the first speculative instruction and determines whether the instruction exception was detected during execution of the first speculative instruction.

52. The computer readable medium of claim 51, wherein the program further includes recovery code to which control is passed when the check instruction determines that the instruction exception was detected during execution of the first speculative instruction.

53. The computer readable medium of claim 51, wherein the program further includes recovery code to which control is passed when the check instruction determines that the instruction exception was detected during execution of the second speculative instruction.

54. The computer readable medium of claim 52, wherein the recovery code includes a copy of the first speculative instruction.

55. A computer readable medium encoded with a compiler that, when executed on a computer system, performs a method of compiling a source program to generate a compiled program, the method comprising a step of:

(A) generating a first speculative instruction that is capable of experiencing an instruction exception condition during execution of the first speculative instruction, wherein the first speculative instruction defers signaling an instruction exception when the exception condition is initially detected and completes execution without signaling the instruction exception.

56. The computer readable medium of claim 55, wherein the first speculative instruction has a first destination, and wherein the step (A) includes a step of generating the first speculative instruction so that when the exception condition is initially detected, the first speculative instruction stores information in the first destination indicating that the instruction exception was detected during execution of the first speculative instruction.

57. The computer readable medium of claim 56, wherein the method further includes a step of generating a second speculative instruction that operates upon a result in the first destination, wherein the second speculative instruction has a second destination, and wherein when the information indicating that the instruction exception was detected during execution of the first speculative instruction is stored in the first destination, the second speculative instruction stores information in the second destination indicating that the instruction exception was detected during execution of the program.

58. The computer readable medium of claim 57. wherein the method further includes a step of generating a check instruction that determines whether the instruction exception was detected during execution of the first speculative instruction.

59. The computer readable medium of claim 58, wherein the method further includes a step of generating recovery code to which control is passed when the check instruction determines that the instruction exception was detected during execution of the first speculative instruction.

60. A computer system, comprising:

means for executing a first program instruction based upon a speculation, the first program instruction being capable of experiencing an instruction exception condition during execution of the first program instruction;

means for deferring signaling of an instruction exception when the exception condition is detected during execution of the first program instruction;

means for determining whether the speculation was incorrect;

means for ignoring the instruction exception when the speculation was incorrect; and

means for changing control flow of the program to recovery code when it is determined that the exception condition was detected during execution of the first program instruction and that the speculation was correct.

61. The computer system of claim 60, further including means for signaling the instruction exception when the speculation was correct.

62. A computer readable medium encoded with a program that, when executed on a computer system, performs a method including steps of:

executing a first program instruction based upon a speculation, the first program instruction being capable of experiencing an instruction exception condition during execution of the first program instruction;

deferring signaling of an instruction exception when the exception condition is initially detected during execution of the first program instruction;

determining whether the speculation was incorrect;

ignoring the instruction exception when the speculation was incorrect; and

changing control flow of the program to recovery code when it is determined that the exception condition was detected during execution of the first program instruction and that the speculation was correct.

63. The computer readable medium of claim 62, wherein the method further includes a step of signaling the instruction exception when the speculation was correct.

64. A computer readable medium having a program stored thereon in a computer-readable form, the program comprising:

a first speculative instruction that receives an operand and is capable of experiencing an instruction exception condition, that when the first speculative instruction receives a first type of operand, executes and calculates a result, and when the first speculative instruction receives a second type of operand including an exception token, simply passes the token to the destination of the speculative instruction without executing the instruction and wherein the first speculative instruction thereby defers signaling an instruction exception when the exception condition is initially detected.

65. A method for determining whether a first instruction and a second instruction related to the first instruction access a common memory location, the method comprising steps of:

executing the first instruction;

uniquely identifying the first instruction in an entry in an address table;

executing the second instruction, wherein executing the second instruction includes searching the address table for an entry identifying the first instruction;

removing the entry identifying the first instruction from the address table when a range of memory space of the first instruction overlaps a range of memory space of the second instruction; and

executing a check instruction that determines whether a common memory location was accessed by the first and second instructions, wherein the check instruction searches the address table for an entry corresponding to the first instruction and determines that a common memory location was accessed by the first and second instructions when an entry for the first instruction is not found in the address table.

66. The method of claim 65, wherein the step of executing a check instruction includes branching to recovery code when a common memory location is accessed.

67. The method of claim 65, wherein the step of executing a check instruction includes determining that a common memory location was not accessed, when an entry for the first instruction is found.

68. A computer system comprising:

a memory;

an address table;

a compiled program stored in a computer-readable form in the memory, the program including:

a store instruction;

a load instruction that is scheduled before the store instruction; and

a check instruction that accesses the address table to determine whether the store instruction and the load instruction access a common location in memory, wherein after the execution of the store instruction an absence of an entry identifying the load instruction in the address table indicates that a common location in memory was accessed by the store and load instructions.

69. The computer system of claim 68, wherein the address table includes a number of entries corresponding to the number of registers used for the load instructions.

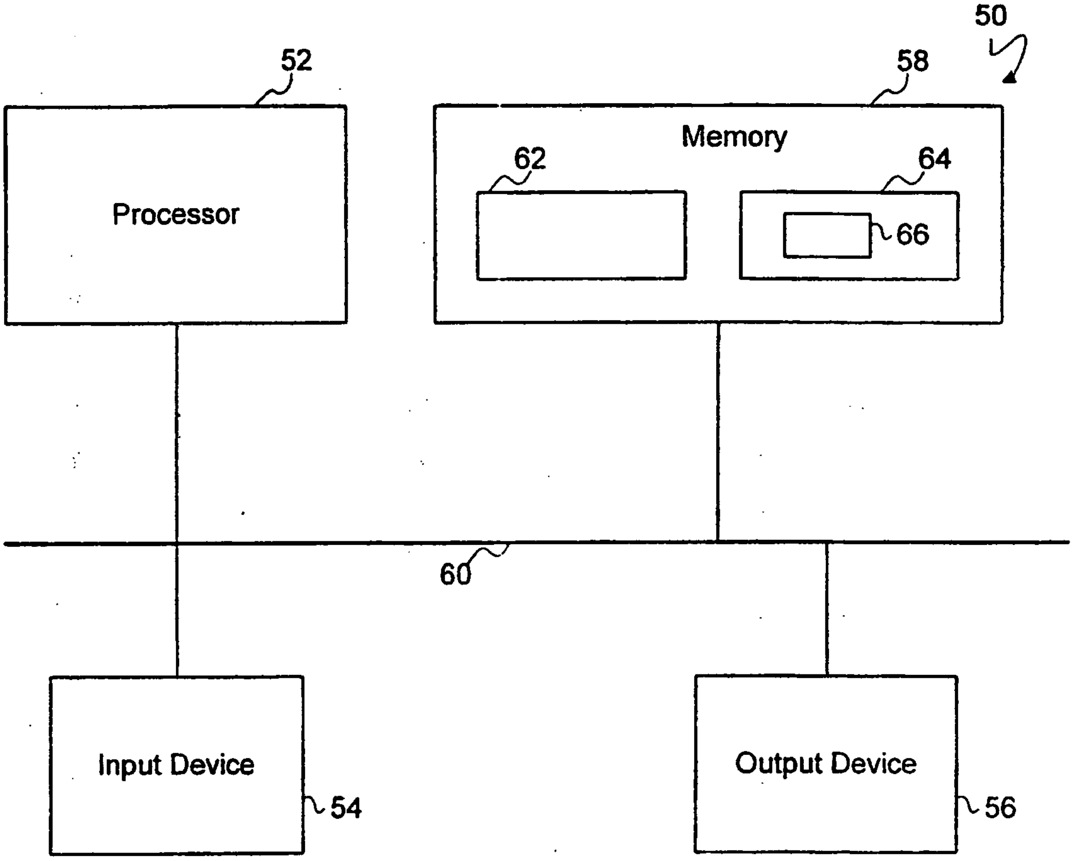
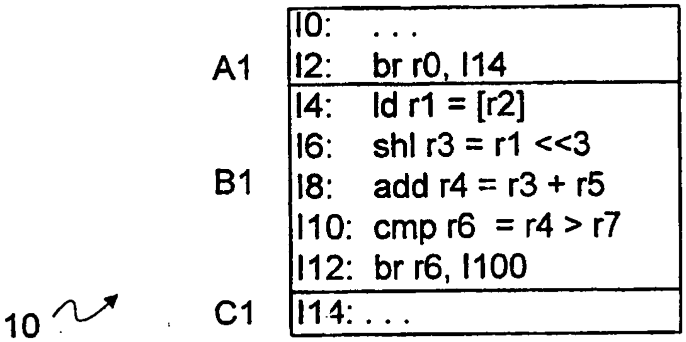
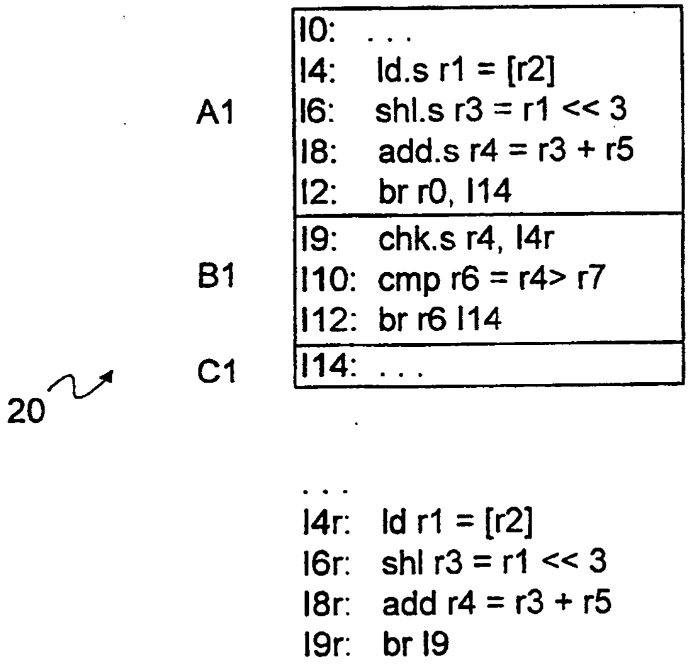

50
52
58
Processor
Memory
62
64
66
60
Input Device
54
Output Device
56

FIGURE 1



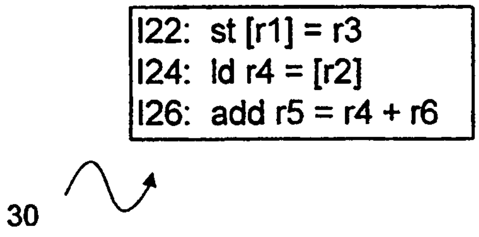
```
I0:   . . .
I2:   br r0, I14
I4:   ld r1 = [r2]
I6:   shl r3 = r1 <<3
I8:   add r4 = r3 + r5
I10:  cmp r6  = r4 > r7
I12:  br r6, I100
I14: . . .
```

FIGURE 2
ORIGINAL CODE



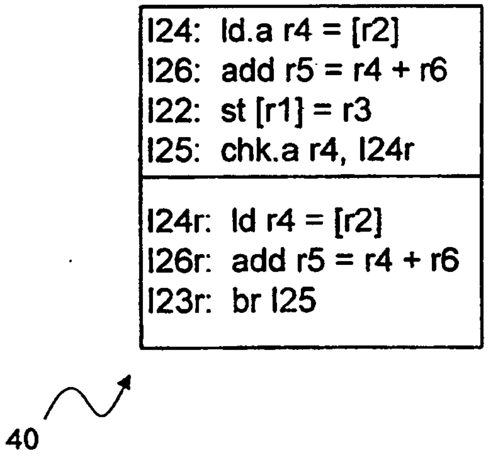
```
I0:   . . .
I4:   ld.s r1 = [r2]
I6:   shl.s r3 = r1 << 3
I8:   add.s r4 = r3 + r5
I2:   br r0, I14
I9:   chk.s r4, I4r
I10:  cmp r6 = r4> r7
I12:  br r6 I14
I14: . . .

. . .
I4r:  ld r1 = [r2]
I6r:  shl r3 = r1 << 3
I8r:  add r4 = r3 + r5
I9r:  br I9
```

FIGURE 3
SCHEDULED CODE WITH
OUT-OF-LINE RECOVERY



```
I22:  st [r1] = r3
I24:  ld r4 = [r2]
I26:  add r5 = r4 + r6
```

FIGURE 4
ORIGINAL CODE



```
I24:  ld.a r4 = [r2]
I26:  add r5 = r4 + r6
I22:  st [r1] = r3
I25:  chk.a r4, I24r

I24r:  ld r4 = [r2]
I26r:  add r5 = r4 + r6
I23r:  br I25
```

FIGURE 5
SCHEDULED CODE WITH
OUT-OF-LINE RECOVERY

FIGURE 6

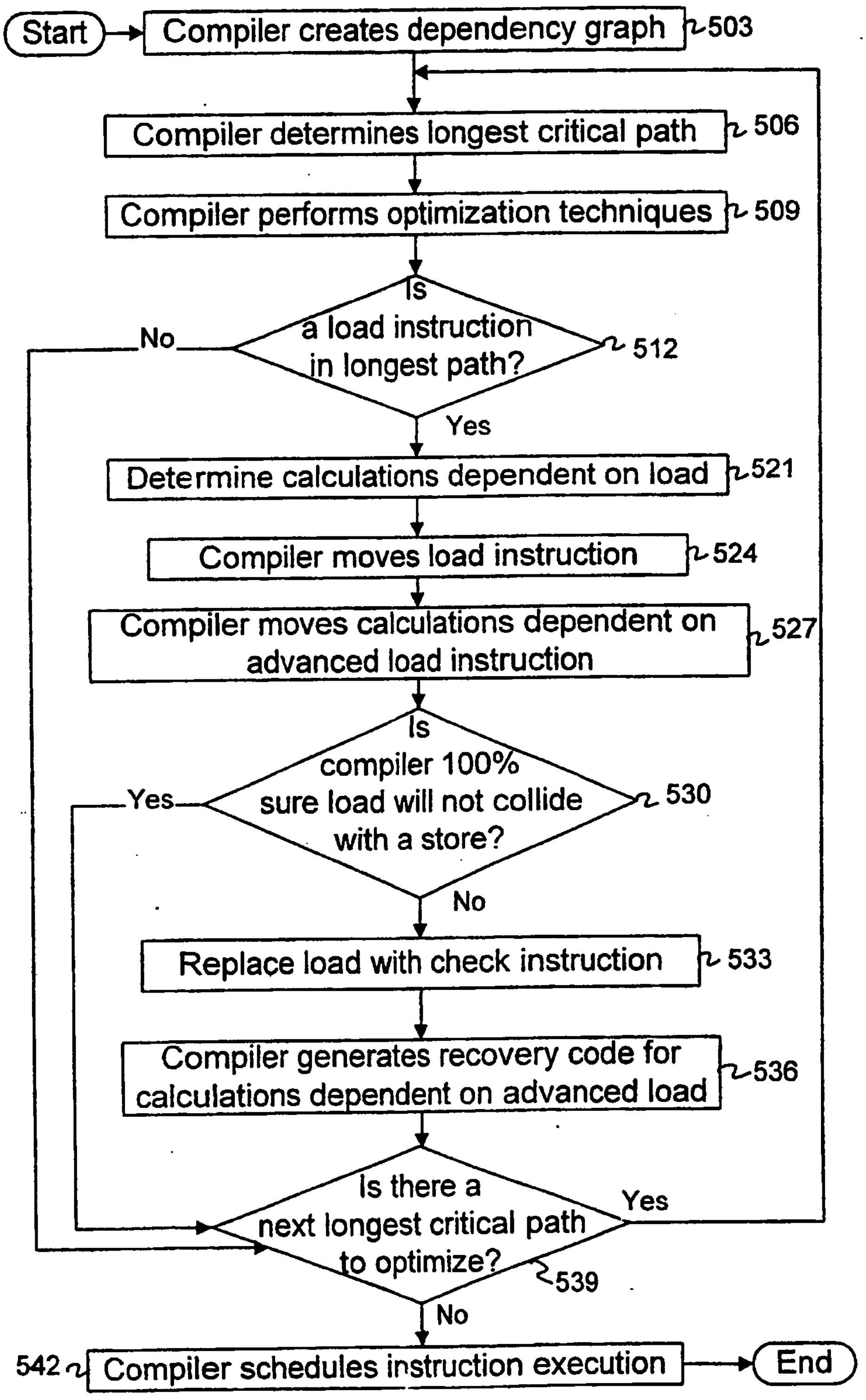
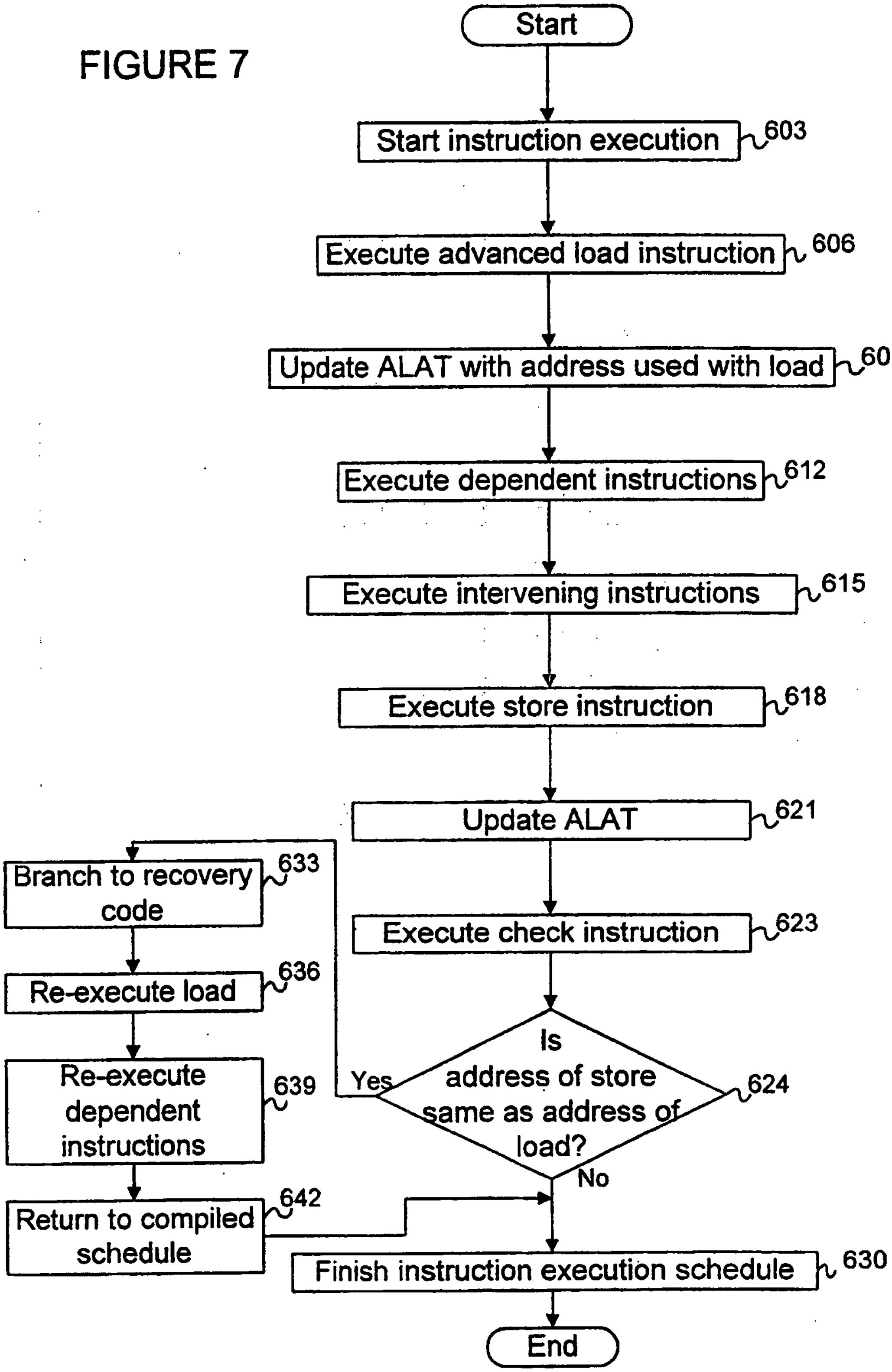

Start
Compiler creates dependency graph
503
Compiler determines longest critical path
506
Compiler performs optimization techniques
509
Is a load instruction in longest path?
512
No
Yes
Determine calculations dependent on load
521
Compiler moves load instruction
524
Compiler moves calculations dependent on advanced load instruction
527
Is compiler 100% sure load will not collide with a store?
530
Yes
No
Replace load with check instruction
533
Compiler generates recovery code for calculations dependent on advanced load
536
Is there a next longest critical path to optimize?
539
Yes
No
542
Compiler schedules instruction execution
End

FIGURE 7


Start
Start instruction execution
603
Execute advanced load instruction
606
Update ALAT with address used with load
609
Execute dependent instructions
612
Execute intervening instructions
615
Execute store instruction
618
Update ALAT
621
Execute check instruction
623
Is address of store same as address of load?
624
Yes
No
Branch to recovery code
633
Re-execute load
636
Re-execute dependent instructions
639
Return to compiled schedule
642
Finish instruction execution schedule
630
End

Advanced Load Address Table (ALAT)				
Memory Address	Memory Access Size	Register Number	Register Type	Valid Bit

FIGURE 8

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US98/21465

A. CLASSIFICATION OF SUBJECT MATTER

IPC(6) :G06F 9/38, 9/45, 11/28
US CL :395/591, 706, 800.23, 800.24
According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

U.S. : 395/591, 706, 800.23, 800.24

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

APS

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	US 5,625,835 A (EBCIOGLU ET AL.) 29 APRIL 1997, SEE ENTIRE DOCUMENT.	1-33 AND 44-69
XP	US 5,778,219 A (AMERSON ET AL.) 07 JULY 1998, SEE ENTIRE DOCUMENT.	34-43

□ Further documents are listed in the continuation of Box C. □ See patent family annex.

*	Special categories of cited documents	"T"	later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"A"	document defining the general state of the art which is not considered to be of particular relevance	"X"	document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"E"	earlier document published on or after the international filing date	"Y"	document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
"L"	document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)	"&"	document member of the same patent family
"O"	document referring to an oral disclosure, use, exhibition or other means		
"P"	document published prior to the international filing date but later than the priority date claimed		

Date of the actual completion of the international search

15 JANUARY 1999

Date of mailing of the international search report

02 APR 1999

Name and mailing address of the ISA/US
Commissioner of Patents and Trademarks
Box PCT
Washington, D.C. 20231
Facsimile No. (703) 305-3230

Authorized officer
WILLIAM M. TREAT
Telephone No. (703) 305-9699

REVISED VERSION*

PCT

WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification ^{6} **:** **G06F 9/38**	**A1**	**(11) International Publication Number:** **WO 99/19795** **(43) International Publication Date:** 22 April 1999 (22.04.99)

(21) International Application Number: PCT/US98/21465

(22) International Filing Date: 9 October 1998 (09.10.98)

(30) Priority Data:

08/953,836	13 October 1997 (13.10.97)	US
09/168,040	7 October 1998 (07.10.98)	US

(71) Applicant: INSTITUTE FOR THE DEVELOPMENT OF EMERGING ARCHITECTURES, L.L.C. [US/US]; c/o Hewlett-Packard Company, Legal Dept. M/S 44L18, 19111 Pruneridge Avenue, Cupertino, CA 95014-0795 (US).

(72) Inventors: MORRIS, Dale, C.; 442 Gilbert Avenue, Menlo Park, CA 94025 (US). MILLS, Jack, D.; 1768 Chevalier Drive, San Jose, CA 95124 (US). CHEN, William, Y.; 1477 Yukon Drive, Sunnyvale, CA 94087 (US).

(74) Agent: HAGGARD, Alan, H.; Hewlett-Packard Company, Legal Dept., M/S 20BN, P.O. Box 10301, Palo Alto, CA 94303-0890 (US).

(81) Designated States: AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CU, CZ, DE, DK, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, UA, UG, UZ, VN, YU, ZW, ARIPO patent (GH, GM, KE, LS, MW, SD, SZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

Published

With a revised version of the international search report.

(88) Date of publication of the revised version of the international search report: 28 September 2000 (28.09.00)

(54) Title: METHOD AND APPARATUS FOR OPTIMIZING EXECUTION OF LOAD AND STORE INSTRUCTIONS

(57) Abstract

Problems encountered during speculative execution of instructions are deferred. If the results of instructions that were speculatively executed are subsequently used, the integrity of the execution of the instructions is verified. If not, recovery code is executed that alters the state of the computer system (50) to create the appearance that the speculatively executed instructions had executed successfully.

*(Referred to in PCT Gazette No. 39/2000, Section II)

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania
AM	Armenia
AT	Austria
AU	Australia
AZ	Azerbaijan
BA	Bosnia and Herzegovina
BB	Barbados
BE	Belgium
BF	Burkina Faso
BG	Bulgaria
BJ	Benin
BR	Brazil
BY	Belarus
CA	Canada
CF	Central African Republic
CG	Congo
CH	Switzerland
CI	Côte d'Ivoire
CM	Cameroon
CN	China
CU	Cuba
CZ	Czech Republic
DE	Germany
DK	Denmark
EE	Estonia
ES	Spain
FI	Finland
FR	France
GA	Gabon
GB	United Kingdom
GE	Georgia
GH	Ghana
GN	Guinea
GR	Greece
HU	Hungary
IE	Ireland
IL	Israel
IS	Iceland
IT	Italy
JP	Japan
KE	Kenya
KG	Kyrgyzstan
KP	Democratic People's Republic of Korea
KR	Republic of Korea
KZ	Kazakstan
LC	Saint Lucia
LI	Liechtenstein
LK	Sri Lanka
LR	Liberia
LS	Lesotho
LT	Lithuania
LU	Luxembourg
LV	Latvia
MC	Monaco
MD	Republic of Moldova
MG	Madagascar
MK	The former Yugoslav Republic of Macedonia
ML	Mali
MN	Mongolia
MR	Mauritania
MW	Malawi
MX	Mexico
NE	Niger
NL	Netherlands
NO	Norway
NZ	New Zealand
PL	Poland
PT	Portugal
RO	Romania
RU	Russian Federation
SD	Sudan
SE	Sweden
SG	Singapore
SI	Slovenia
SK	Slovakia
SN	Senegal
SZ	Swaziland
TD	Chad
TG	Togo
TJ	Tajikistan
TM	Turkmenistan
TR	Turkey
TT	Trinidad and Tobago
UA	Ukraine
UG	Uganda
US	United States of America
UZ	Uzbekistan
VN	Viet Nam
YU	Yugoslavia
ZW	Zimbabwe

REVISED VERSION

INTERNATIONAL SEARCH REPORT

In ational Application No
PCT/US 98/21465

A. CLASSIFICATION OF SUBJECT MATTER
IPC 6 G06F9/38

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 6 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	EP 0 742 512 A (IBM) 13 November 1996 (1996-11-13) column 2, line 9 - line 44 column 4, line 27 - line 48 column 6, line 13 -column 7, line 8 column 7, line 29 - line 51 column 8, line 50 -column 9, line 33 column 7, line 57 -column 12, line 24 column 12, line 39 -column 13, line 28 --- -/--	1-16, 18-33, 65-69

☒ Further documents are listed in the continuation of box C.

☒ Patent family members are listed in annex.

° Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance
"E" earlier document but published on or after the international filing date
"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)
"O" document referring to an oral disclosure, use, exhibition or other means
"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.
"&" document member of the same patent family

Date of the actual completion of the international search 23 June 2000	Date of mailing of the international search report 04.07.00
Name and mailing address of the ISA European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+31-70) 340-3016	Authorized officer Daskalakis, T

2

INTERNATIONAL SEARCH REPORT

In ational Application No
PCT/US 98/21465

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication,where appropriate, of the relevant passages	Relevant to claim No.
E	US 5 903 749 A (KARP ALAN ET AL) 11 May 1999 (1999-05-11) the whole document	1-16, 18-33, 65-69
P,X	& WO 98 00769 A (IDEA CORP) 8 January 1998 (1998-01-08) the whole document ---	1-16, 18-33, 65-69
X	GALLAGHER D M ET AL: "DYNAMIC MEMORY DISAMBIGUATION USING THE MEMORY CONFLICT BUFFER" ACM SIGPLAN NOTICES,US,ASSOCIATION FOR COMPUTING MACHINERY, NEW YORK, vol. 29, no. 11, 1 November 1994 (1994-11-01), pages 183-193, XP000491733 ISSN: 0362-1340 the whole document ---	1-16, 18-33, 65-69
A	EP 0 568 842 A (IBM) 10 November 1993 (1993-11-10) column 12, line 38 -column 13, line 39 column 15, line 26 -column 17, line 41 ---	1-16, 18-33, 65-69
X	EP 0 789 298 A (HEWLETT PACKARD CO) 13 August 1997 (1997-08-13) the whole document ---	34-64
X	WO 96 23254 A (IBM) 1 August 1996 (1996-08-01) the whole document ---	34-37, 39-42, 44,46, 48-53, 55-64
P,X	US 5 692 169 A (WORLEY JR WILLIAM S ET AL) 25 November 1997 (1997-11-25) the whole document -----	34-64

International application No.
PCT/US 98/21465

INTERNATIONAL SEARCH REPORT

Box I Observations where certain claims were found unsearchable (Continuation of item 1 of first sheet)

This International Search Report has not been established in respect of certain claims under Article 17(2)(a) for the following reasons:

1. □ Claims Nos.:
because they relate to subject matter not required to be searched by this Authority, namely:

2. □ Claims Nos.:
because they relate to parts of the International Application that do not comply with the prescribed requirements to such an extent that no meaningful International Search can be carried out, specifically:

3. □ Claims Nos.:
because they are dependent claims and are not drafted in accordance with the second and third sentences of Rule 6.4(a).

Box II Observations where unity of invention is lacking (Continuation of item 2 of first sheet)

This International Searching Authority found multiple inventions in this international application, as follows:

see additional sheet

1. □ As all required additional search fees were timely paid by the applicant, this International Search Report covers all searchable claims.

2. □ As all searchable claims could be searched without effort justifying an additional fee, this Authority did not invite payment of any additional fee.

3. ☒ As only some of the required additional search fees were timely paid by the applicant, this International Search Report covers only those claims for which fees were paid, specifically claims Nos.:

1-16,18-69

4. □ No required additional search fees were timely paid by the applicant. Consequently, this International Search Report is restricted to the invention first mentioned in the claims; it is covered by claims Nos.:

Remark on Protest

□ The additional search fees were accompanied by the applicant's protest.

☒ No protest accompanied the payment of additional search fees.

FURTHER INFORMATION CONTINUED FROM PCT/ISA/ 210

This International Searching Authority found multiple (groups of) inventions in this international application, as follows:

1. Claims: 1-16, 18-33, 65-69

   Computer system and method to schedule and execute load instructions before conceptually preceding stores.

2. Claim : 17

   Computer system for scheduling execution of load and store instructions to different execution units.

3. Claims: 34-64

   Computer system and method to speculatively execute instructions that may generate exceptions.

INTERNATIONAL SEARCH REPORT

Information on patent family members

Int ional Application No
PCT/US 98/21465

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
EP 0742512 A	13-11-1996	US 5625835 A	29-04-1997
		JP 8314721 A	29-11-1996
US 5903749 A	11-05-1999	AU 3644797 A	21-01-1998
		WO 9800769 A	08-01-1998
EP 0568842 A	10-11-1993	JP 2786574 B	13-08-1998
		JP 6214799 A	05-08-1994
		US 5542075 A	30-07-1996
EP 0789298 A	13-08-1997	US 5778219 A	07-07-1998
		JP 9223017 A	26-08-1997
WO 9623254 A	01-08-1996	US 5799179 A	25-08-1998
		CN 1136182 A	20-11-1996
		CZ 9702084 A	17-12-1997
		DE 69600995 D	24-12-1998
		DE 69600995 T	08-07-1999
		EP 0804759 A	05-11-1997
		JP 8263287 A	11-10-1996
		PL 321542 A	08-12-1997
US 5692169 A	25-11-1997	DE 19534752 A	25-04-1996
		GB 2294341 A	24-04-1996
		JP 8123685 A	17-05-1996
		US 5778219 A	07-07-1998
		JP 5241864 A	21-09-1993